

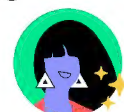


Spring Boot em Ação

Parafraseando um livro muito conhecido sobre o framework Spring Boot, veremos ao longo desse módulo, tal ferramenta em ação. Para isso, construiremos uma API Restful composta pelas camadas Entity, Repository, Service e Controller. Além disso, veremos detalhes complementares de implementação, como o arquivo de propriedades do projeto e também o gerenciamento de dependências utilizando o Maven.

Antes de começarmos, você precisará ter em seu computador uma IDE (qualquer uma de sua preferência – eu sugiro a Spring Tool Suite). Caso queira visualizar a estrutura de bancos de dados do projeto (utilizaremos o banco de dados embarcado H2), instale também uma ferramenta como a DBeaver. Além disso, você também precisará da máquina virtual do Java. Sugiro que instale o JDK na versão 11. Todos esses softwares são gratuitos. Com tais ferramentas já instaladas, vamos dar início ao nosso projeto.

Para iniciarmos um projeto Spring, podemos utilizar a própria IDE ou, então (e, nesse caso, preferencialmente) um recurso disponibilizado pelo próprio Spring, o site start.spring.io. Ao acessar o site por um navegador web, você visualizará a página demonstrada abaixo, na Figura 1 (na figura, já estará com as opções preenchidas e selecionadas). Antes de prosseguirmos, vamos entender essas opções disponíveis para criação de nosso projeto:



- Project: aqui você escolherá o gerenciador de dependências do projeto. Deixe a opção Maven Project selecionada (ou, caso já

tenha familiaridade com o Gradle, escolha tal opção);



- Language: linguagens disponíveis. Deixa marcada a opção Java;

- Spring Boot: versões do framework. Deixe marcada a opção default, ou seja, 2.7.3 .

- Project Metadata: as informações inseridas em Group e Artifact comporão o Package name de nosso projeto. Na prática, o caminho das pastas de nosso código. Há alguns padrões de mercado para preenchimento dessas informações, como iniciar o Group com "com" , "br.com" ou "org", por exemplo. A seguir, depois do ".com" ou "org", é comum ser inserido o nome da empresa ou organização responsável pelo projeto. Já em Artifact é comum inserirmos o nome do projeto. Em descrição você pode inserir uma descrição breve.

- Packaging: selecione Jar;

- Java: selecione a versão da máquina virtual java instalada em seu computador. Caso tenha instalado a versão 11, conforme sugerido acima, marque aqui a opção correspondente;

- Dependencies (do lado direito): aqui podemos selecionar as dependências (bibliotecas) que utilizaremos em nosso projeto. Há várias opções disponíveis. Nesse primeiro projeto, escolhe as mesmas dependências mostradas na Figura 1.

Após ter realizado as configurações acima, clique no botão "GENERATE", ao final da página. Tal ação fará com que a estrutura do projeto seja baixada para o seu computador. A partir daqui,

poderemos importar o projeto na IDE e começamos, então, a codificar.



spring initializr

Project: Maven Project / Gradle Project
Language: Java / Kotlin / Groovy
Spring Boot: 3.0.0 (SNAPSHOT) / 3.0.0 (M4) / 2.7.4 (SNAPSHOT) / 2.7.3 / 2.6.12 (SNAPSHOT) / 2.6.11

Project Metadata
Group: br.com.descomplica
Artifact: projeto
Name: projeto
Description: API Restful com Spring Boot
Package name: br.com.descomplica.projeto
Packaging: Jar / War
Java: 18 / 17 / 11 / 8

Dependencies — ADD DEPENDENCIES... CTRL + B

Spring Boot DevTools — DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Spring Web — WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA — SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database — SQL
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

GENERATE CTRL + ⏎ | EXPLORE CTRL + SPACE | SHARE...

Figura 1: Página inicial do Spring initializr

O processo de importação do projeto varia conforme a IDE que você estiver utilizando. Para mais detalhes, há vários tutoriais e vídeos disponíveis na internet.

O primeiro passo, após importação do projeto na IDE, consiste na criação dos “packages” para organização das camadas de nossa API. A Partir do package principal, você vai criar (se usou o package name sugerido na Figura 1, o mesmo será br.com.descomplica.projeto) os pacotes: entity, repository, service e controller. Com o projeto organizado e estruturado, em termos de packages (ou pastas/diretórios, caso tenha criado os mesmos a partir do sistema

de arquivos, no seu sistema operacional), vamos começar a, efetivamente, codificar. Embora não haja um ponto certo a partir do qual começar, sugiro iniciarmos pelas Entidades. Logo, vamos realizar o mapeamento objeto relacional das tabelas e relações constantes no DER exibido a seguir, na Figura 2:

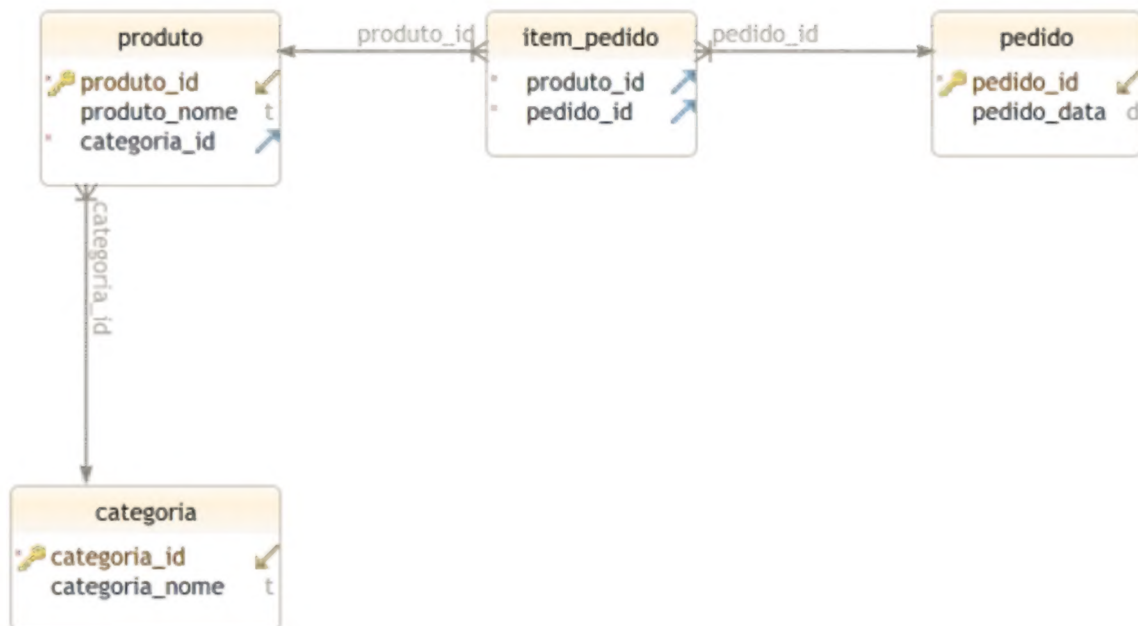


Figura 2: DER do Projeto

A partir do DER, podemos criar no package Entity, as classes correspondentes a cada tabela do banco, com exceção da tabela de ligação, "item_pedido". Por se tratar de uma tabela de ligação que não possui colunas extras (possuindo apenas as chaves-primárias das tabelas relacionadas), tal tabela não precisará ser mapeada em uma entidade própria.

Nota: sugiro que você tente criar os códigos, sem olhar as respostas que serão disponibilizadas mais adiante, ao final desse módulo.

Seguem alguns lembretes:



- Inclua as anotações de classe e também as de atributos e relacionamentos;

- Em relação às anotações de relacionamento, analise o DER para entender a cardinalidade entre cada Entidade;

- Crie os métodos get e set (as IDEs costumam ter um atalho para geração desses métodos).

Depois de criar as entidades, seus atributos e também relacionamentos, podemos criar as interfaces no package Repository. Nesse sentido, crie uma interface para cada Entidade. Lembre-se de que, por uma questão de boas práticas, o nome de cada repositório deverá ser formado pelo nome da entidade seguido do sufixo Repository. Além disso, você deverá estender a interface JpaRepository. Nesse ponto, uma informação adicional: ao estender a JpaRepository, você deverá fornecer dois parâmetros dentro dos sinais “<>”, a saber: o nome da entidade e o tipo de dado com o qual você declarou o atributo que representa a chave-primária na entidade. Por exemplo: caso você tenha criado a Entidade “Produto” com um atributo chamado produtoId e declarar seu tipo como Integer, ao estender a JpaRepository, seu código ficará assim: “... extends JpaRepository<Produto, Integer>...”.

A criação das interfaces do pacote repository é a parte mais simples em nossa API – considerando que, na mesma, implementamos apenas os métodos CRUD. Após finalizada essa etapa, vamos criar os nossos serviços.

Os serviços no Spring são classes Java anotados com @Service. Essas classes são responsáveis por manter as regras de negócio. Além disso, elas fazem a intermediação entre o Controller e o Repository. Mantendo a convenção indicada, deveremos criar um Service para cada Entidade – no padrão "NomedaEntidadeService.java". Antes de continuarmos, precisamos falar sobre a forma como o Spring cuida da Injeção de Dependências(DI) e da Inversão de Controle(IoC). Normalmente, caberia ao programador tratar tais questões em seu código. Entretanto, o Spring possui uma forma bastante simples de fazê-lo, assumindo em nosso lugar tal função. Para instanciarmos e utilizarmos, ao longo das camadas do Spring, classes (e seus recursos públicos) pertencentes a outra camada, basta as definirmos utilizando a anotação @Autowired (pertencente ao pacote org.springframework.beans.factory.annotation.Autowired). Com isso, o Spring se encarrega de instanciar a classe e de disponibilizá-la para nosso uso. Vejamos alguns exemplos:

a) Estando no ProdutoController, precisaremos acessar a classe ProdutoService. Para isso, na Classe ProdutoController, defina uma instância do Service e a anote, dessa forma:

@Autowired

ProdutoService produtoService;

Após esse ponto, poderemos utilizar os recursos do Service, dentro do controller, com "produtoService.NOME-DO-RECURSO";

b) A exemplo do que fizemos acima, no Controller, podemos também fazer no Service (no exemplo, em

ProdutoService), onde acessaremos o Repository:



@Autowired

ProdutoRepository produtoRepository;

Dando continuidade ao nosso projeto, agora você já pode criar todas as classes Service.

Na hierarquia de nosso projeto, por fim, chegamos até a camada Controller. Essa camada conterá os recursos que disponibilizaremos em nossa API – em nosso projeto, os métodos CRUD. O papel dos métodos contidos no Controller é apenas expor os recursos, endereços (URI) e formas de acesso (mapeamento do verbo HTTP) aos mesmos, direcionando as requisições recebidas para a camada inferior, a Service. Logo, nessa camada não deverão constar regras de negócio ou acesso direto à camada Repository (o Spring não o impede de realizar esse acesso, mas é fortemente recomendado seguir a hierarquia proposta pelo mesmo, em que cada camada deve acessar apenas a camada imediatamente inferior ou superior). O Controller, além de receber as requisições de fora de nossa API, também é responsável por respondê-las. Nesse sentido, podemos também manipular os códigos/status HTTP de cada resposta, utilizando as classes “ResponseEntity” e “HttpStatus” (ambas do pacote org.springframework.http).

Agora, crie um Controller para cada Entidade, utilizando os padrões de nomenclatura aplicados até aqui. Lembre-se, também, de anotar cada Classe Controller com as anotações @RestController e @RequestMapping (essa última para definir um patch específico para o recurso. Por ex: /produto; /pedido; etc.). Além disso, cada método CRUD constante na classe deve ser anotado com o verbo HTTP através do qual será acessado.

Veja, a seguir, os códigos prontos de todas as camadas. A seguir, veremos como configurar nosso projeto para acessar o banco de dados. Então, poderemos testar nossa API.

Link:
https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/tree/main/modulo4/codigo_fonte **(Acesso em 14/09/2022)**

Uma parte muito importante do Spring Boot é o arquivo de propriedades – criado automaticamente, ao gerarmos o projeto pelo start.spring.io , chamado “application.properties” e que fica localizado na pasta “src/main/resources”. Esse arquivo, que também pode ter o formato “yaml”, é responsável por armazenar uma série de informações da aplicação, como o tipo e credenciais do banco de dados a ser utilizado, as configurações de log, segurança, entre outras. Nele podemos inserir, além de propriedades definidas pelo Spring, Hibernate, etc., propriedades customizadas, como as credenciais de uma API externa que acessamos, endereço de pastas para armazenamento de arquivos, entre outras. A seguir, você poderá ver as propriedades de nossa API. Algumas delas são autoexplicativas. Para mais informações ou em caso de dúvidas, consulte a documentação oficial do Spring.

```
#na config abaixo, os dados serao perdidos cada vez que a API for reiniciada
#spring.datasource.url=jdbc:h2:mem:projetodb
#na config abaixo, os dados sao gravados em arquivo e, com isso, nao serao perdidos qdo a API for reiniciada
spring.datasource.url=jdbc:h2:file:/data/projetodb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true

#a propriedade abaixo forca o spring a executar, sempre que subir a API, o script data.sql
#  a propriedade so devera ficar descomentada na primeira vez que subir a API ou quando o arquivo data.sql for alterado
#spring.sql.init.mode=always
spring.jpa.defer-datasource-initialization=true

#a propriedade abaixo informa que a estrutura do banco devera ser verificada, a partir das Entidades, e atualizada qdo a API
iniciar
spring.jpa.hibernate.ddl-auto=update

#as propriedades abaixo servem para exibir, no console da IDE, informacoes detalhadas sobre a execucao de queries pela
API
#logging.level.org.hibernate.SQL=debug
#logging.level.org.hibernate.type.descriptor.sql=trace
```

Arquivo de Propriedades

Repare que há algumas tralhas (símbolo #) no arquivo. Tal símbolo é utilizado para a inserção de comentários, não sendo reconhecido como código a ser executado. Nos comentários foram inseridas algumas explicações sobre as propriedades utilizadas, para ajudá-lo na compreensão das mesmas.

Por fim, para testar nossa API, faremos uso de ferramentas externas. Entre as opções disponíveis, duas se destacam: Postman e Insomnia. Através dessas ferramentas podemos testar APIs do tipo Restful. Após instalar uma dessas opções, crie requisições, de acordo com a URI, métodos/endpoints e anotações de verbo HTTP constantes na aplicação. Cada requisição, se executada corretamente, devolverá a resposta no formato JSON (formato de dados padrão em APIs do tipo Rest) e também o código do status HTTP correspondente – conforme definimos em nosso código. Para

facilitar o processo de testes, importe a coleção abaixo para o Insomnia (ou converta-a para utilização no Postman).

Link: https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/tree/main/modulo4/insomnia_collection **(Acesso em 14/09/2022)**

Ao longo desse módulo, codificamos uma API Restful utilizando o Spring Boot. Com isso, podemos entender melhor as funcionalidades e recursos disponíveis em tal framework.

Atividade Extra

É comum encontrarmos alguns problemas de recursividade no Spring, no momento de recuperarmos dados de entidades que possuem relacionamentos entre si. Se tiver interesse em saber mais sobre recursividade e como tratar tal questão, veja o link a seguir:

Link: https://www.baeldung.com/jackson-bidirectional-relationships-and-infinite-recursion . **(Acesso em 14/09/2022)**

Referência Bibliográfica

WALLS, Craig. **Spring Boot in Action**. Shelter Island, NY: Manning Manning, 2016.



Atividade Prática Módulo 4

Título da Prática: Criando a camada Service.

Objetivos: Codificar a camada Service de uma API Restful.

Materiais, Métodos e Ferramentas: IDE (Spring Tool Suite; Jetbrains Intellij; etc)

Atividade Prática

Chegou a hora de criarmos a camada Service de nossa API. Tal camada é responsável por armazenar os serviços, cujas responsabilidades são: guardar as regras de negócio e fazer a intermediação entre as camadas Controller e Repository. Considerando as instruções abaixo, crie as Classes Services correspondentes às entidades em questão.

- Entidades contidas na API: Instrutor e Turma
- Tipo de dado dos atributos @Id de ambas as entidades: Integer
- No Controller estão disponíveis dois métodos, para os quais deverão ser criados dois métodos no Service:

    procurarPodId: receberá um inteiro como parâmetro e devolverá a instância da entidade correspondente ao id fornecido;

salvar: receberá uma nova instância da entidade a ser armazenada no banco de dados;



- Os códigos das entidades, repositórios e controles estão disponíveis a seguir:

```
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import com.fasterxml.jackson.annotation.JsonBackReference;
import com.fasterxml.jackson.annotation.JsonIdentityInfo;
import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.ObjectIdGenerators;

@Entity
@Table(name = "turma")
public class Turma {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id_turma")
private Integer idTurma;

@Column(name = "horario")
private Date horarioTurma;

@Column(name = "duracao")
private Integer duracaoTurma;
```

```
import java.util.Date;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
import javax.validation.constraints.NotEmpty;

import com.fasterxml.jackson.annotation.JsonIdentityInfo;
import com.fasterxml.jackson.annotation.ObjectIdGenerators;

@Entity
@Table(name = "instrutor")
public class Instrutor {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id_instrutor")
private Integer idInstrutor;

@Column(name = "rg")
private Integer rgInstrutor;

@Column(name = "nome")
private String nomeInstrutor;

@Column(name = "nascimento")
private Date dataNascimento;

@Column(name = "titulacao")
private Integer titulacaoInstrutor;

@OneToMany(mappedBy = "instrutor")
private List<Turma> turmaList;
```

- Repository

```
import org.springframework.data.jpa.repository.JpaRepository;

//import *.entity.Turma;

public interface TurmaRepository extends JpaRepository<Turma,Integer> {
}
```



```
import org.springframework.data.jpa.repository.JpaRepository;
//import *.entity.Instrutor;

public interface InstrutorRepository extends
JpaRepository<Instrutor,Integer> {
}
```

- Controller



```
//imports ...

@RestController
@RequestMapping("/turma")
public class TurmaController {

@GetMapping("/{id}")
public Turma procurarPorId(@PathVariable Integer id) {
      //aqui deverão ser implementados, posteriormente, os códigos para
localizar por id
      //return Turma
}

@PostMapping
public Turma salvar(@RequestBody Turma turma) {
      //aqui deverão ser implementados, posteriormente, os códigos para
salvar a Entidade
      //return Turma;
}
}
```

```
//imports ...

@RestController
@RequestMapping("/instrutor")
public class instrutorController {

@GetMapping("/{id}")
public Instrutor procurarPorId(@PathVariable Integer id) {
      //aqui deverão ser implementados, posteriormente, os códigos para
localizar por id
      //return Instrutor
}

@PostMapping
public Instrutor salvar(@RequestBody Instrutor instrutor) {
      //aqui deverão ser implementados, posteriormente, os códigos para
salvar a Entidade
      //return Instrutor;
}
}
```

Gabarito Atividade Prática

Os códigos disponibilizados a seguir representam a resposta da atividade em questão. Nesse sentido, as respostas desse gabarito representam o código mínimo a ser entregue pelo aluno. Logo, as

respostas poderão variar, mas precisarão conter, no mínimo, o que é exposto abaixo.



```
//imports ...

@Service
public class TurmaService {
@Autowired
TurmaRepository turmaRepository;

public Instrutor procurarPorId(Integer id) {
      //aqui deverão ser implementados, posteriormente, os códigos para
localizar por id
      //return Instrutor
}

public Instrutor salvar(Instrutor instrutor) {
      //aqui deverão ser implementados, posteriormente, os códigos para
salvar a Entidade
      //return Instrutor;
}
}
```

```
//imports ...

@Service
public class InstrutorService {
@Autowired
InstrutorRepository instrutorRepository;

public Instrutor procurarPorId(Integer id) {
      //aqui deverão ser implementados, posteriormente, os códigos para
localizar por id
      //return Instrutor
}

public Instrutor salvar(Instrutor instrutor) {
      //aqui deverão ser implementados, posteriormente, os códigos para
salvar a Entidade
      //return Instrutor;
}
}
```

Ir para exercício